# Modular Composition of Language Features through Extensions of Semantic Language Models

Claus Pahl
School of Computer Applications, Dublin City University
Dublin, Ireland

**Abstract**

Today, programming or specification languages are often extended in order to customize them for a particular application domain or to refine the language definition. The extension of a semantic model is often at the centre of such an extension. We will present a framework for linking basic and extended models. The example which we are going to use is the RSL concurrency model. The RAISE specification language RSL is a formal wide-spectrum specification language which integrates different features, such as state-basedness, concurrency and modules. The concurrency features of RSL are based on a refinement of a classical denotational model for process algebras. A modification was necessary to integrate state-based features into the basic model in order to meet requirements in the design of RSL. We will investigate this integration, formalising the relationship between the basic model and the adapted version in a rigorous way. The result will be a modular composition of the basic process model and new language features, such as state-based features or input/output.

We will show general mechanisms for integration of new features into a language by extending language models in a structured, modular way. In particular, we will concentrate on the preservation of properties of the basic model in these extensions.

## 1 Integration through Extension

The specification and development of complex software systems might require the use of different specification features, assembled into a customised language. The combination of specification features requires a thorough understanding of all particular features involved. In this paper, we will present a framework which supports the systematic, modular integration of denotationally defined formal models of features by stepwise, property-preserving extension of the language semantics.

Modularity and compositionality in the design, integration, and customisation of languages are key issues which are not yet solved. This was pointed out in the group reports and position statements of the ACM Workshop on Strategic Directions in Computing Research, MIT, June 1996, in particular the groups Programming Languages and Software Engineering and Programming Languages, which were partially published in ACM Computing Surveys [ACM96]. Semantics of languages is about descriptions of computational features. These computational features should be described separately and then assembled to more comprehensive languages. [Hoa96, HJ98] also refer to the extension of programming languages through inclusion of new language features, such as variables or procedures: 'an essential goal is to manage such newly introduced complexity by use of as much as possible of the existing theory and algebra. Ideally, each new feature can be defined and introduced separately, in a way that permits them to be combined without further complexities of interaction.'

We will present principles and a case study using an extension calculus and notation geared towards modular, composable descriptions of languages based on extensions of their semantical models. This paper is based on [Pah98], which introduces the ideas used here in a more general way. A formal model defining a language feature possesses some properties: functions behave in a certain way, semantic domains might be constrained. We will show how these qualitatively different properties can be preserved. The practical advantage of preserving properties can be illustrated by considering the behaviour of programs implemented on the concurrency model. It can be guaranteed that a program preserves its behaviour if it is executed as a program of the extended language. The behaviour is not

necessarily identical since underlying semantical notions might have changed, but there exists an observation under which the behaviour is identical.

The formal specification framework RAISE [Geo91, Gro92] is an example of a combination of different specification techniques. RSL is the RAISE specification language. RSL is based on principles of VDM [Jon90], but also includes features for the specification of concurrency and a modularity concept. Bolignano and Debabi's paper [BD92] is the basis for this investigation. They describe the extension of a simple denotational model for CSP-like process algebras based on the acceptances model [Hen85] to meet the requirements in the development of RAISE. We will follow [BD92] and investigate their development of a refined model in a more rigorous and formal way considering aspects of modularity in design and extension of language models or languages. This will result in a modular, comprehensible description of the integration of formal models. To illustrate our approach, we will add state-based, model-theoretic features – such as input/output via channels or states and values – to the basic model of processes. Processes might depend on some external events such as the communication via channels. In a simple model, the events can be left unspecified, input/output might not be dealt with. In a modular, stepwise development, the notion of events can then be refined. New commands can be added, existing ones have to be redefined in order to work on new structural requirements. We will provide a framework in which these extensions are formalised by using operators in an algebraic framework. We will provide a selection of common templates of extension which support conservative, property-preserving extensions. By providing such templates, we will facilitate the process of adapting domain and function definitions of a basic model to meet new requirements.

We will start with introducing the basic language in Section 2. The following section 3 introduces principles of our approach of extension. Behaviour and domain constraints are important properties which are addressed. We use these results in Section 4 for a first extension by partitioning events. A second extension in Section 5 adds states and values. We conclude with related work.

## 2 Basic Model and Language

The basic model of processes, which we are going to introduce, is based on Hennessy's acceptances model [Hen85], but in the representation of Bolignano and Debabi [BD92]. The acceptances model was introduced as a denotational model for CSP-like process algebras[1].

**LANGUAGE DESCRIPTION**

**Syntax**

*Process ::= 'stop' | 'chaos' | Event '→' Process | Process [] Process*

**Semantic Domain**

$P_0 = ((\Sigma \to P_0) \times \mathcal{P}\mathcal{P}\Sigma)_\perp$

**Semantic Functions**

$$
\begin{array}{lcl}
\mathbf{P_0} & : & Process \to P_0 \\
\mathbf{P_0}[\![chaos]\!] & \triangleq & \perp \\
\mathbf{P_0}[\![stop]\!] & \triangleq & ([\,], \{\emptyset\}) \\
\mathbf{P_0}[\![e \to p]\!] & \triangleq & ([e \mapsto \mathbf{P_0}[\![p]\!]], \{\{e\}\}) \\
\mathbf{P_0}[\![p_1 [] p_2]\!] & \triangleq & \text{let } (M_1, S_1) = \mathbf{P_0}[\![p_1]\!], (M_2, S_2) = \mathbf{P_0}[\![p_2]\!] \text{ in } (M_1 []_m M_2, S_1 []_s S_2) \\
with & & \\
M_1 []_m M_2 & \triangleq & (([e \mapsto M_1(e) | e \in dom(M_1)] \triangle [e \mapsto M_2(e) | e \in dom(M_2)]) \sqcup \\
 & & \quad [e \mapsto M_1(e) []_m M_2(e) \mid e \in dom(M_1) \cap dom(M_2)]) \\
S_1 []_s S_2 & \triangleq & \{x | x \in \mathcal{P}\Sigma \wedge x \subseteq \bigcup \{y | y \in S_1 \cup S_2\} \wedge \exists v \in S_1, w \in S_2 . x \subseteq v \cup w\}
\end{array}
$$

Figure 1: The basic language definition

[1] A similar extension of the acceptances model can be found in [HI93b, HI93a] where a simple process language is extended by values and assignments.

A process domain will be the kernel structure containing acceptance sets as one of its parts. A simple process language based on an acceptances model is presented in figure 1[2].

A sample process description is

$$(a \rightarrow stop)[](b \rightarrow c \rightarrow stop)$$

This process can choose externally (choice is determined by the environment) between two events $a$ and $b$. If $a$ is chosen it stops. If $b$ is chosen, the process can engage in another event $c$ before it stops. Later on, we introduce a constraint that guarantees that a process can only engage in a certain set of events (the acceptance set) in each process state.

The **basic domain**, which is the main semantic domain of the language, is a space of processes $P_0$

$$P_0 = ((\Sigma \rightarrow P_0) \times \mathcal{PP}\Sigma)_\perp$$

where $\Sigma$ is a set of events. The first component of those pairs is a mapping from events to processes in $P_0$. The second component is an acceptance set – some constraints on the domain $\mathcal{PP}\Sigma$ will be introduced later. An acceptance set contains all events that a process can engage in. We could say that such a set stands for the possible internal states that can be reached. Each of these states is a set of actions that can be taken in that particular state. $\perp$ is the semantic correspondence to the $chaos$ process. The domain is defined recursively, but a solution exists, see [BD92] section 5. [BD92] introduces two nondeterministic choice operators. Only the external choice operator $[]$ shall be considered here, the internal choice will be neglected in order to simplify the presentation. Other language elements denote a deadlock process, $stop$, a divergent process, $chaos$, and a process which depends on an external event, $e \rightarrow p$.

Bolignano and Debabi impose semantical **constraints on the process domain** which guarantee consistency of the definition and also specify properties of acceptance sets:

1. Events in process descriptions have to be acceptable, i.e. for $(m, S) \in P_0$:

   $$dom(m) = \bigcup S$$

2. Acceptance sets have to be saturated. Saturation subsumes closure conditions on sets. $A$ has to be a *saturated* finite subset of $\mathcal{P}\Sigma$, expressed by $Sat(A)$, for $A \in \mathcal{PP}\Sigma$.

   $$Sat(A) \triangleq \forall x \in A, y \in \mathcal{P}\Sigma.(y \subseteq \bigcup\{z | z \in A\} \wedge x \subseteq y) \Rightarrow y \in A$$

   This is an adaption of criterion $Sat$ in Figure 3 in [BD92]. The reason for using $\{z|z \in A\}$ instead of $A$ can be seen in subsequent reformulations of this definition. For subsets $A_i$ of a power set, $\bigcup A_i \in A$ is the union closure, and $S_1, S_2 \in A \ \wedge \ S_1 \subseteq T \subseteq S_2 \ \Rightarrow \ T \in A$ is the convex closure.

The consistency of the language definition with the constraints has to be proven, e.g. it has to be shown for the external choice $[]$:

1. $dom(M_1[]_m M_2) = dom(M_1) \cup dom(M_2)$ with $dom(M_1) = \bigcup S_1$ and $dom(M_2) = \bigcup S_2$, i.e. $dom(M_1[]_m M_2) = \bigcup S_1 \cup \bigcup S_2 = \bigcup(S_1 \cup S_2)$. $S_1[]_s S_2 \subseteq \bigcup(S_1 \cup S_2)$ by construction, $x \subseteq \bigcup\{y \mid y \in S_1 \cup S_2\}$ and thus $S_1[]_s S_2 = \bigcup(S_1 \cup S_2)$, $\exists v \in S_1, w \in S_2.x \subseteq v \cup w$. Thus, we have equality.

2. $x \in S_1[]_s S_2$ is saturated obviously by definition of $Sat$.

The consistency of the basic language is not an issue of the extension approach, but has rather to be resolved by the language designer for a particular language description.

[2] We use VDM operators such as symmetric difference $\triangle$, map extension $\sqcup$, or override $\dagger$ in the figures.

# 3 Principles of Extension

In the previous section, we have presented a small process description language with a denotational concurrency model at its core. This model shall be extended: the abstract communication events shall be made more concrete as input/output events, values (to be passed through input channels) and internal states (to store these values) shall be introduced. The necessary formal framework for model extensions will be introduced in this section.

The principles of behaviour preserving and domain constraint preserving extensions will be motivated. The basic construct, a semantics extension, will be introduced. It plays the role of an extension operator which is equipped with extension laws characterising the properties to be preserved. We will show that domain constraints can technically be dealt with as a special form of behaviour.

## 3.1 Motivation

Principles of extension shall be motivated using the syntactical phrase $e \rightarrow p$ as a primitive process description whose properties have to be preserved in an extension of the process model. The process domain shall be extended in this example by adding a value domain. This example is not part of the actual RSL-model extension, it is only used to illustrate extensions using a simple and common new feature. An injection shall be used to refine the process domain $P_0 = (\Sigma \rightarrow P_0) \times \mathcal{PP}\Sigma$ into a product of values, processes and acceptance sets. Let us start with the semantic function defining the language phrase $e \rightarrow p$:

$$\mathbf{P_0}[\![e \rightarrow p]\!] \stackrel{\triangle}{=} ([e \mapsto \mathbf{P_0}[\![p]\!]], \{\{e\}\})$$

$e \rightarrow p$ is a phrase of the syntactic domain $Process$. The semantic function $\mathbf{P_0}$ maps elements from $Process$ to $P_0$. The event $e$ is associated with the meaning of the process description $p$. The event $e$ has to be an acceptable event.

Let us now consider an extension of the language which contains again the phrase $e \rightarrow p$. Processes shall be extended by allowing them to have a value in each process state. Thus, the semantic function has to be adapted by introducing a value domain $V$. This is expressed using a type constructor $T$ which maps the basic process domain $P_0$ into a product of values and processes $TP_0$:

$$T : P_0 = ((\Sigma \rightarrow P_0) \times \mathcal{PP}\Sigma)_\perp \;\; \mapsto \;\; TP_0 = (V \times (\Sigma \rightarrow TP_0) \times \mathcal{PP}\Sigma)_\perp$$

We will now briefly introduce constructs needed to describe an extension. They will be explained in detail in the subsequent subsections. A mapping $\phi : P_0 \rightarrow TP_0$ maps elements of the process domain into the extension. The mapping $\_^*$ is a function lifting which extends the semantic function $\mathbf{P_0}$ to the new structural requirements given by the extended domain $TP_0$, i.e. $\_^* : \mathbf{P_0} \mapsto \mathbf{P_0^*}$.

The previous paragraphs have illustrated the extension of the description of process behaviour. The given domain constraint, expressing e.g. that processes can only engage in acceptable events,

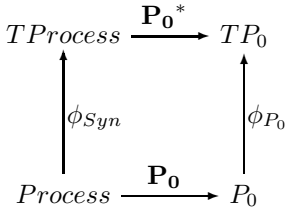
$$dom(m) = \bigcup S \;\text{ for } (m, S) \in P_0$$

are expressed on the basic domain $P_0$. Extending $P_0$ to $TP_0$, we also have to extend the constraints. In particular, we have to preserve the intended restriction of behaviour. We will show a standard way of rewriting a constraint such that the constraint is preserved.

## 3.2 Behaviour Preserving Extensions

### *3.2.1 Behaviour Preservation*

The semantic function $\mathbf{P_0}^*$ preserves certainly the behaviour of $\mathbf{P_0}$, if the following diagram commutes:




or textually expressed by

$$\mathbf{P_0}^*[\![\phi_{Syn}(e \to p)]\!] \;=\; \phi_{P_0} \circ \mathbf{P_0}[\![e \to p]\!]$$

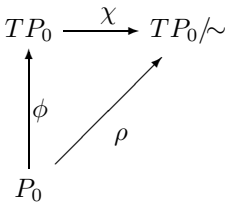
This is mathematically the homomorphism criterion on algebras. We have used a morphism $\phi_{Syn}$ on the syntactical domain $Process$ which shall denote an embedding of $Process$ into the extended syntactical domain $TProcess$. It is not straightforward here to talk about the behaviour of processes which has to be preserved. The semantics for processes is here not given in an operational semantics style, e.g. in terms of a transition system. The semantics of a process is given here as a semantic entity which, by its recursive structure, describes the possible future process states, depending on events. Our aim is to preserve these semantics in an extension.

The notion of behaviour preservation indicates that behavioural aspects of the basic model should reappear in the extension. The homomorphism criterion might be too restrictive. We introduce a more flexible observability criterion applicable to the extension, which explains what has to be preserved. We can use an equivalence $\sim$ as the observation criterion instead of equality:

$$\mathbf{P_0}^*[\![\phi^{\sim}_{Syn}(e \to p)]\!] \;\sim\; \phi^{\sim}_{P_0} \circ \mathbf{P_0}[\![e \to p]\!]$$

$\phi$ describes data refinement. $\phi^{\sim}$ is derived from $\phi$ by mapping elements into equivalence classes of an equivalence $\sim$ on the extended domain $TP_0$. The equivalence explains which elements in the extended domain represent the same element from the basic domain. The extension $\mathbf{P}_0^*$ has to be defined in accordance with $\phi^{\sim}$, if behaviour has to be preserved.

Processes $p = (ep, s)$ from $P_0$ are refined to $(v, ep, s)$ in $TP_0$ for some value $v$. Some extended elements are observably equivalent, expressed by an equivalence class $[(v, ep, s)]_{\sim}$. The relation between basic and extended domain is expressed by a mapping $r : P_0 \to TP_0/\!\sim$ where $[(v, ep, s)]_{\sim} := \{(a, b, c) \mid b = ep \wedge c = s\}$. We have used an equivalence $\sim$ on $TP_0$ to define relevant observable behaviour. Extended functions are expected to behave correctly with respect to the equivalence classes. A mapping $c : (v, ep, s) \mapsto [(v, ep, s)]_{\sim}$ does always exist, $q : (ep, s) \mapsto (v, ep, s)$ can be constructed. The set-based mappings such as $r, p$ or $c$ can be extended to corresponding morphisms $\rho, \phi, \chi$ on algebras based on congruences and quotient algebras such that $\chi \circ \phi = \rho$ with $\phi : P_0 \to P_0^*$, $\chi : P_0^* \to P_0^*/\!\sim$ and $\rho : P_0 \to P_0^*/\!\sim$.




$\phi$ is a morphism which preserves behaviour and $\chi$ is a canonical morphism onto the quotient algebra. If $\sim$ is a congruence, then $\chi$ is a canonical morphism, i.e. always exists; see [Coh65, Grä68] for details.

### *3.2.2 Semantics Extensions*

The previous introduction of concepts shall now be summarised in the definition of a semantics extension, the basic building block of our extension approach. This construct is an extension operator which will be equipped with extension laws describing the properties to be preserved. Applied to an algebra which models a basic language, we can obtain another, extended algebra which interprets the extended language. Semantics extensions describe a language feature in separation.

**Definition 3.1** *Let $\mathcal{A}$ and $\mathcal{B}$ be algebras, let $A$ and $B$ be sets in $\mathcal{A}$ and $\mathcal{B}$. A* **semantics extension** *from $\mathcal{A}$ to $\mathcal{B}$ is a 5-tuple $(T, \phi^\sim, \sim, \_^*, \delta)$ where*

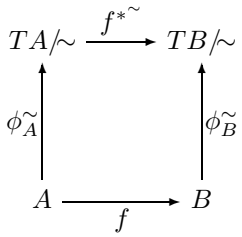
- *$T : \mathcal{A} \mapsto \mathcal{B}$ is a collection of type constructors on algebras,*
- *$\phi^\sim : \mathcal{A} \mapsto \mathcal{B}/\!\sim$ is a collection of mappings on sets,*
- *$\sim$ is a binary relation on $TA$ if $A$ is set of $\mathcal{A}$,*
- *$\_^*$ is a function lifting, which lifts $f : A \to B$ to $f^* : TA \to TB$,*
- *$\delta_\sim : TA/\!\sim \to TA$ is a collection of typed choice operators, yielding default values for each equivalence class in $TA/\!\sim$.*

The equality on quotients is the equivalence on the basic sets. $\phi^\sim$ can be extended to a homomorphisms on algebras. For the following discussion, we often assume a domain (a set) of interest $S$. Then, for all domains $D$ not equal to $S$, $\sim_D$ is assumed to be the equality and $\phi_D$ is assumed to be the identity mapping.

Constraints on semantics extensions are called **extension laws**. They describe the property to be preserved.

**Definition 3.2** *Assume a semantics extension $(T, \phi^\sim, \sim, \_^*, \delta)$ and an arbitrary function $f : A \to B$ in a basic algebra. Let $f^{*^\sim} := [f^*]_\sim$. The* **behaviour preservation extension law** *is defined as $f^{*^\sim} \circ \phi^\sim = \phi^\sim \circ f$.*

We can express the law diagrammatically as:




**Definition 3.3** *Assume a semantics extension $(T, \phi^\sim, \sim, \_^*, \delta)$.*

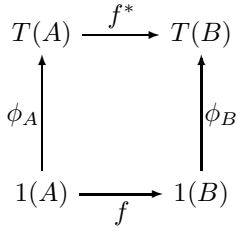
- *The extension is called* **faithful**, *if for any two functions $f, g : A \to B$ of the same type holds:*

$$f \neq g \;\Rightarrow\; f^{*^\sim} \neq g^{*^\sim}$$

- *The extension is called* **full**, *if $\_^{*^\sim}$ is a surjective mapping from $A \to B$ to $TA/\!\sim \to TB/\!\sim$ for every combination of $A$ and $B$.*

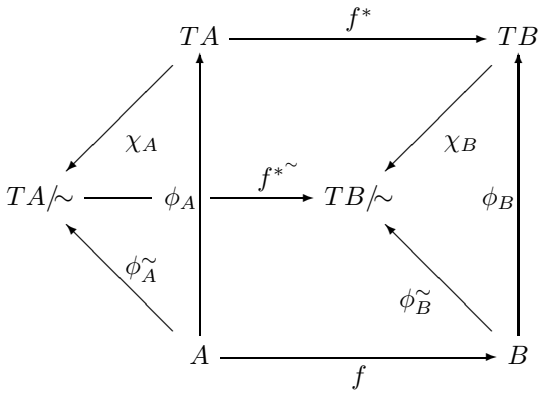
If functions are behaviourally distinguishable on the basic layer, they should not be *observably* different on the extension layer. This is formalised by the notion of faithfulness. The equivalence $\sim$ defines observationally equivalent behaviour. For each class of extended functions, there should be the corresponding original function. Full extensions reflect this issue. For the remainder we expect extensions to be full and faithful.

In terms of category theory, $\phi$ is a natural transformation with respect to the endofunctors 1 and $T$, if




where $T(f) := f^*$. A natural transformation preserves structure and behaviour. As explained above, this is a too restrictive constraint. We have introduced equivalence classes to relax that above condition.

We shall now summarise these results in a single diagram which illustrates how to construct an extension from a semantics extension:




The behaviour preservation law is encoded in the leftward-slanted rectangle at the bottom. The extension itself is described by the square. We can always derive $\phi$ such that $\chi$ exists. Only now, we need the defaults $\delta$ to construct the morphism $\phi$. Defaults allow us to pick a particular elements from equivalence classes to construct $\phi$ from $\phi^\sim$. The resulting mapping $\phi$ is not necessarily a homomorphism. Note, that in the diagram, not all combinations of arrows are commuting based on equality, sometimes it is only equivalence. A semantics extension is a construct similar to Kleisli triples $(T, \eta, \_^*)$ from category theory [Mog91], where $T$ is a type constructor, $\eta$ an embedding into the domain constructed by $T$, and $\_^*$ is a function lifting (slightly different from ours). To point out the similarity, we have used a similar notation. However, our extension is more general due to the introduction of equivalences. We will come back to Kleisli triples later on.

### *3.2.3 Extension Templates*

We now investigate how a semantics extension can be constructed easily such that the extension laws are satisfied.

**Definition 3.4** *An* **extension template** *is a semantics extension which satisfies the extension laws.*

We will look at the idea of templates in the context of behaviour preservation. Let $S \mapsto TS$ be the domain extension. An equivalence $\sim$ or a mapping $\phi^\sim$ for $TS$ cannot be derived automatically from $S$ in general, but it is possible for some cases based on particular domain extensions. There is a standard way of obtaining a behaviour preserving function lifting.

**Proposition 3.1** *The function lifting $\_^*$, which lifts a function $f : A \to B$ to $f^* : TA \to TB$ for $a \in A$, defined by*

$$f^*(\phi_A(a)) := \phi_B(f(a))$$

*guarantees behaviour preservation for a semantics extension $(T, \phi^\sim, \sim, \_^*, \delta)$.*

**Proof:** The definition of $f^*$ is partial, but total on the relevant subset of $TA$. This definition guarantees $f^*(\phi_A(a)) = \phi_B(f(a))$, i.e. equality as a particular equivalence. □

**Definition 3.5** *We will define* **extension templates**, *based on a particular domain extension. We will give* $\sim, \phi^\sim, \_^*, \delta_\sim$ *for each* $T$ *on a domain* $S$.

1. $T : S \mapsto S \times R$ **:** $(s,r) \sim (s',r')$ *iff* $s = s'$; $\phi^\sim : s \mapsto [(s,r)]_\sim$; $\delta([(s,r_0)]) = (s,r_0)$
2. $T : S \mapsto S + R$ **:** $x \sim x'$ *iff* $x = x'$; $\phi^\sim : s \mapsto [s]$; $\delta([s]) = s$
3. $T : S \mapsto (I \to S)$ **:** $t \sim t'$ *iff* $\forall i \in I.t(i) = t'(i)$; $\phi^\sim : s \mapsto t$ *with* $t(i) = s$ *for all* $i \in I$; $\delta([t]) = \lambda i.s$
4. $T : S \mapsto \mathcal{P}(S)$ **:** $p \sim p'$ *iff* $p = p'$; $\phi^\sim : s \mapsto [\{s\}]$; $\delta([\{s\}]) = \{s\}$

All templates are behaviour preserving semantics extensions based on predefined constructions for type extension, e.g. for injection $S \mapsto S \times T$ (see figure 2 for a formulation in our extension notation) or indexing $S \mapsto (I \to S)$ (figure 4). The templates have to satisfy a number of constraints: the equivalence $\sim$ is a congruence, there is a default value for each equivalence class, the function lifting satisfies the behaviour preservation law.

**Proposition 3.2** *The templates are behaviour preserving semantics extensions, i.e. they satisfy the behaviour preservation extension law.*

**Proof:** The well-formedness of the template components for the four templates is easy to see, classical injections or embeddings are used. Straightforward with proposition 3.1. □

**Proposition 3.3** *The extension templates define full and faithful extensions.*

**Proof:** As it can be seen from the construction of $\sim$ and $\phi^\sim$, we have isomorphy between a set $A$ and the quotient set $TA/\sim$. From that it follows immediately that the templates define full and faithful semantics extensions. □

**EXTENSION TEMPLATE**

INJECT $S$ INTO $S \times R$ $=$ ( $T : S \mapsto S \times R$
$\_^* : f_i \overset{c}{\mapsto} f_i^*$
$(s,r) \sim (s',r')$ iff $s = s'$
$\phi_S^\sim(s) = [(s,r_0)]_\sim$
$\delta : [(s,r_0)]_\sim \mapsto (s,r_0)$ )

Figure 2: Extension template INJECT

The idea behind templates is to reduce the amount of information that a language designer has to give for the application of an extension operator. The domain type extension $T$ is essential and has to be chosen by the language designer explicitly, but then we can use canonical ways of defining an extension operator. Templates can form a library of extension operators for the language designer.

## 3.3 Domain Constraint Preserving Extensions

We have addressed the preservation of behavioural properties in the previous subsection. Let us now consider the preservation of structural domain constraints. An example of such a domain constraint is the constraint which defines that a process can only engage in acceptable events. Constraints on domains can be interpreted in two ways: as a construction of a subdomain or as a property that has to be preserved. We will pursue the second alternative. It is a more general approach and does not interfere with behaviour preservation. Thus, domain constraints are properties similar to behaviour, also characterised by an extension law. They have to be preserved by appropriate extension templates.

### *3.3.1 Domain Constraint Preservation*

**Definition 3.6** *A* **domain constraint** *is a predicate* $P : A \to Bool$ *on a domain* $A$*.*

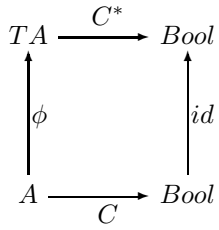
Domain constraints are expressed as predicates. Two examples were presented in section 2 for the process domain $P_0$. The extension of domains is expressed by the type construction $T : S \mapsto TS$. Elements of domains are mapped by $\phi : S \to TS$ into the extension. We will start our investigation assuming a semantics extension $(T, \phi^\sim, \sim, \_^*, \delta)$ with a type constructor $T$, an extension mapping $\phi^\sim$, and an overloaded lifting operator $\_^*$, called **domain constraint lifting** when applied to a domain constraint.

**Definition 3.7** *The* **domain constraint preservation law** *is satisfied, if*

$$C^* \circ \phi = id \circ C$$

*for a domain constraint predicate* $C$ *on a domain* $A$ *and a domain constraint lifting* $\_^*$*.*

This means that $C^*$ has to hold iff $C$ holds. The introduction of $id$ will become clear when we analyse domain and behaviour preservation together. The law can be expressed diagrammatically:




An example is $dom(m) = \bigcup S$, which is a predicate $C$ on $P_0 = ((\Sigma \to P_0) \times \mathcal{PP}\Sigma)_\perp$, i.e. $C(P_0) := dom(m) = \bigcup S$ for all $(m, S) \in P_0$. The predicate could as well be a binary relation or might involve other domains. Predicates are implicitly universally quantified.

Domain constraints are essentially specific forms of behavioural specification. Thus, the domain constraint preservation law is a specific behaviour preservation law. For domain constraints, $\phi_{Bool}$ is the identity and $\sim$ is just equality.

### *3.3.2 Extension Templates*

A template for extending a domain constraint, such that the domain constraint preservation law is satisfied, shall now be introduced.

**Definition 3.8** *Let* $(T, \phi^\sim, \sim, \_^*, \delta)$ *be a semantics extension. A domain constraint* $C$ *on a domain* $S$ *is extended to* $C^*$ *on* $TS$*, called a* **constraint extension template***, as follows. Substitute syntactically each application of variable* $s_i$ $(i = 1, .., n)$ *in the constraint* $C$ *by fresh variables* $s_i'$ *and each occurrence of domain* $S$ *by* $TS$ *in quantifications. Then, the inverse* $\phi^{-1}$ *for* $\phi : S \to TS$ *is applied to the elements* $s_i'$*, i.e. substitute syntactically* $s_i'$ *by* $\phi^{-1}(s_i')$*. Thus, we get:*

$$C^* := C[s_1/\phi^{-1}(s_1'), \ldots, s_n/\phi^{-1}(s_n'), S/TS]$$

Constraints which are extended by the above *constraint extension template* are automatically satisfied in the extension.

**Proposition 3.4** *The constraint extension template satisfies the constraint extension law.*

**Proof:** Obvious due to construction via inverses. □

**Proposition 3.5** *The constraint extension template defines a full and faithful extension.*

**Proof:** The inverses allow us to construct an isomorphism. Thus, we have a full and faithful semantics extension. □

Let us illustrate the template using an example. We consider a domain $S$ which is extended by injection to $S \times T$. Let there be the constraint $\forall m \in S \,.\, dom(m) = \bigcup S$. The constraint is extended adding fresh variables and extending the domain. Since $dom$ is a function on $S$, we will adapt using the inverse. The inverse operation to injection is projection[3]. We get $\bigcup S = \bigcup\{\pi_1(m') \mid m' \in S \times T\}$, and finally $\forall m' \in S \times T \,.\, dom(\pi_1(m')) = \bigcup S$.

The domain extension $S \mapsto TS$ gives rise to the canonical construction of *behaviour templates* as well as *domain templates*. The constraint extension is a general-purpose template applicable to all kinds of domain extension. It lifts the original constraint such that its validity is preserved.

# 4 Partitioning Events

Principles of our extension approach have now been presented in an example and we can start extending the basic model from section 2. In the first step, we partition the set of events into input, output and termination events. Input and output denote the directions of communication via channels. Before we address the concrete problem, we introduce the general technique of partitioning in form of an extension template. The extension of the process domain with behaviour preservation as well as the extension of domain constraints preserving the original constraints will be considered.

## 4.1 Partitioning Template

A domain $S$ might be partitioned into subdomains $S_{k_1}, \ldots, S_{k_n}$, if a function $kind : S \to K$ with $K = \{k_1, \ldots, k_n\}$ exists, which assigns a unique kind to each $s \in S$. The partitioning shall be expressed explicitly through an extension template based on a product domain

$$T : S \mapsto S_{k_1} \times \ldots \times S_{k_n}$$

such that $s \in S_{k_i}$ whenever $kind(s) = k_i$[4]. The partitioning shall constitute the extension, i.e. we have to define the equivalence $\sim$ and the mapping $\phi^{\sim}$. A template shall guarantee the extension laws. Additionally, it should preserve a partitioning on the basic domain $S$ expressed by a function $kind$. The equivalence $\sim$ on $S_{k_1} \times \ldots \times S_{k_n}$ shall be defined by

$$[s_1, \ldots, s_n] \;\sim\; [t_1, \ldots, t_n] \quad \text{iff} \quad kind(s_i) \;=\; kind(t_i) \quad \text{for all } i = 1, .., n$$

The definition of $kind$ induces an equivalence $\sim_P$ on the source domain $S$: all elements of the same kind are equivalent, i.e. for $s, t \in S$: $s \;\sim_P\; t$ iff $kind(s) = kind(t)$. $\phi^{\sim}$ shall be defined by

$$\phi^{\sim} : s \mapsto \text{case } kind(s) \text{ in } k_i \Rightarrow [s_{k_1}, \ldots, s_{k_n}]$$

with $s_{k_i} = s$ and $s_{k_j} = \omega_s$ for $i \neq j$. $\omega_s$ is an undefined value. The behaviour preservation law is satisfied if the canonical function lifting is used. The template is presented in figure 3. This template of partitioning has the existence of $kind$ as a precondition. It also requires the satisfaction of the substitution property, i.e. the equivalence $\sim$ has to be a congruence. The assumed function $kind$ is used to derive an equivalence on $S$ which is used instead of the predefined equality as the behaviour preservation criterion.

## 4.2 Behavioural Extension

The set of events $\Sigma$ shall be partitioned. We assume that we can distinguish input events, output events and a termination event $\surd$ in $\Sigma$. The template PARTITION, or rather a variant of it, shall now be applied. Let $\Sigma_1 = \mathcal{P}\Sigma_{in} \times \mathcal{P}\Sigma_{out} \times \mathcal{P}\Sigma_{\surd}$ be the partitioning of events $\Sigma_1$. The extended process $P_1$ space shall be defined as:

$$P_1 = ((\Sigma_{in} \to P_1) \times (\Sigma_{out} \to P_1) \times \mathcal{P}\mathcal{P}\Sigma_1)_{\perp}$$

[3] Examples for these inverses with respect to the extension operators are injection $d \mapsto (d, e)$ and projection $\pi_1(d, e) = d$ as the retrieval or indexing $d \mapsto f$ with $\forall i.f(i) = d$ and application $f(i)$ as the retrieval.

[4] Choosing a product construction is only one possibility to realise partitioning. Certainly, using a disjoint sum $S \mapsto S_{k_1} + \ldots + S_{k_n}$ is another possibility.

**EXTENSION TEMPLATE**

PARTITION $S$ INTO $S_{k_1} \times \ldots \times S_{k_n} \triangleq$
( $T : S \mapsto S_{k_1} \times \ldots \times S_{k_n}$
$\_^* : f \stackrel{c}{\mapsto} f^*$
$[s_1, \ldots, s_n] \sim'_P [t_1, \ldots, t_n]$ iff $kind(s_i) = kind(t_i)$ for all $i$
$\phi^\sim(s) \stackrel{\triangle}{=} s \mapsto$ case $kind(s)$ in $k_i \Rightarrow [s_{k_1}, \ldots, s_{k_n}]$
with $s_{k_i} = s, s_{k_j} = \delta(\phi^\sim(s))$ for $i \neq j$
$\delta : [(s_1, \ldots, s_n)]_\sim \mapsto (s_1, \ldots, s_n)$ )

**Precondition**: existence of $kind$, substitution property holds

Figure 3: Extension template PARTITION

The following assumptions shall be made. $\Sigma$ can be expressed by a disjoint sum $\Sigma = \Sigma_{in} + \Sigma_{out} + \Sigma_\surd$, i.e. the function $kind$ exists. Functions in the specification, which are supposed to be lifted, preserve the partitioning, i.e. the equivalence $\sim$ on $\Sigma_1$ as a congruence. This can be guaranteed by the canonical construction. Thus, the precondition of the template is satisfied. $\Sigma_\surd$ consist of only one element $\surd$ which denotes immediate termination. We will use a variant of the template which partitions $\Sigma$ into an indexed set in the function space $\Sigma \to P_1$. The variant PART_IND is obtained by applying PARTITION to each first component of maplets of type $\Sigma \to P_1$:

PART_IND $(\Sigma \to P_1)$ INTO $(\Sigma_{in} \to P_1) \times (\Sigma_{out} \to P_1)$

This variant is also behaviour preserving. We get the following definitions for $\phi$ and $\sim$ for the template PART_IND, where the empty map $[\,]$ is used as the default element:

$$
\begin{array}{lcl}
\phi([e \mapsto p]) & = & \text{case } e \text{ in} \\
 & & \quad in(e) : ([e \mapsto p], [\,]) \\
 & & \quad out(e) : ([\,], [e \mapsto p]) \\
 & & \quad \surd(e) : ([\,], [\,]) \\
\phi([\,]) & = & ([\,], [\,]) \\
\phi(\bot) & = & \bot
\end{array}
$$

and

$$
\begin{array}{lll}
(a, b) \sim (a', b') & \text{iff} & a = a' \wedge b = b' \\
[\,] \sim [\,] & \text{iff} & [\,] = [\,]
\end{array}
$$

The following definitions for the lifted semantic function $\mathbf{P_1}$ are derived by application of the template:

$$
\begin{array}{lcl}
\mathbf{P_1} & : & Process \to P_1 \\
\mathbf{P_1}[\![chaos]\!] & \stackrel{\triangle}{=} & \bot \\
\mathbf{P_1}[\![stop]\!] & \stackrel{\triangle}{=} & ([\,], [\,], \{(\emptyset, \emptyset, \emptyset)\}) \\
\mathbf{P_1}[\![e \to p]\!] & \stackrel{\triangle}{=} & \phi(\mathbf{P_0}[\![e \to p]\!]) \\
\mathbf{P_1}[\![p_1 [] p_2]\!] & \stackrel{\triangle}{=} & \text{let } (IN_1, OUT_1, S_1) = \mathbf{P_1}[\![p_1]\!], (IN_2, OUT_2, S_2) = \mathbf{P_1}[\![p_2]\!] \text{ in} \\
 & & \quad (IN_1 []_m IN_2, OUT_1 []_m OUT_2, S_1 []_s S_2)
\end{array}
$$

The formulation of $\mathbf{P_1}[\![p_1 [] p_2]\!]$ as given above does not correspond syntactically directly to the application of the template, but is semantically equivalent and easier to read.

## 4.3 Extending the Domain Constraint

Now, we address the extension and preservation of domain constraints. The constraints, as presented above in section 2 for the basic model, are:

1. $dom(m) = \bigcup S$ for $(m, S) \in P$,
2. $Sat(A)$ for $A \in \mathcal{PP}\Sigma$ with
   $Sat(A) \triangleq \forall x \in A, y \in \mathcal{P}\Sigma.(y \subseteq \bigcup\{z | z \in A\} \wedge x \subseteq y) \Rightarrow y \in A$.

These constraints cannot be applied to the new domain structure. A reformulation is necessary. This reformulation can be done using the domain constraint extension template.

1. For all acceptance sets $S_1$ and $i \in \Sigma_{in}, o \in \Sigma_{out}, \surd \in \Sigma_{\surd}$:

   $$dom(i) = \bigcup\{\pi_1(z) | z \in S_1\} \quad \wedge \quad dom(o) = \bigcup\{\pi_2(z) | z \in S_1\} \quad \wedge$$
   $$dom(\surd) = \bigcup\{\pi_3(z) | z \in S_1\}$$

   with $S_1 = \phi(S)$ and $\Sigma = \Sigma_{in} + \Sigma_{out} + \Sigma_{\surd}$. The projection $\pi$ is the inverse of injection – remember that a product was used to represent the partitioning.

2. $Sat_1(A)$ for $A \in \mathcal{PP}\Sigma_1$ where

   $$\begin{array}{rl} Sat(A) \triangleq & \forall x \in A, y \in (\mathcal{P}\Sigma \times \mathcal{P}\Sigma \times \mathcal{P}\Sigma) \,.\, ( \\ & \quad \pi_1(y) \subseteq \bigcup\{\pi_1(z) | z \in A\} \ \wedge \ \pi_1(x) \subseteq \pi_1(y) \ \wedge \\ & \quad \pi_2(y) \subseteq \bigcup\{\pi_2(z) | z \in A\} \ \wedge \ \pi_2(x) \subseteq \pi_2(y) \ \wedge \\ & \quad \pi_3(y) \subseteq \bigcup\{\pi_3(z) | z \in A\} \ \wedge \ \pi_3(x) \subseteq \pi_3(y) \ ) \\ & \Rightarrow y \in A \end{array}$$

   with $\Sigma_1 = \mathcal{P}\Sigma_{in} \times \mathcal{P}\Sigma_{out} \times \mathcal{P}\Sigma_{\surd}$.

Since the constraint extension template is used, the constraint is preserved. The first part of the constraint is partitioned into three parts and then projections are used to reduce to the original constraint. Using the product for extension and then projecting is also used to obtain the second criterion.

# 5 Values and States

Values and states shall be added to the previous extension in a single step. Values are read via input channels and are bound to variables of the process state. An extension in two steps is therefore not adequate. We will start, as in the previous section, introducing the mechanisms in their general form, before we apply them to the concrete extension construction considering behaviour and domain constraint preservation.

## 5.1 Injection and Indexing Templates

We will need two templates for this extension. The template INJECT was already presented in figure 2. The INDEX template is presented in figure 4. Elements of a basic set $S$ shall be indexed by elements from an index set $I$, i.e. we construct a function space with the basic set as the range. Two extended elements are equivalent, if they map to the same element in the range.

## 5.2 Behavioural Extension

States and values will form a new component of the process space. Values can also be read (*in*) from or written (*out*) onto channels. The resulting domain $P_2$ of the last extension presented here is:

$$P_2 = (\mathcal{P}(S \times V) \times (\Sigma_{in} \to V \to P_2) \times (\Sigma_{out} \to V \to P_2) \times \mathcal{PP}\Sigma_1)_\perp$$

**EXTENSION TEMPLATE**

$$\begin{array}{lll} \text{INDEX } S \text{ BY } I \to S & \triangleq & ( \quad T : S \mapsto I \to S \\ & & \quad \_^* : f \overset{c}{\mapsto} f^* \\ & & \quad f \sim f' \Leftrightarrow f(i) = f'(i) \text{ for all } i \\ & & \quad \phi_S^{\sim}(s) \triangleq [f^s]_\sim \text{ with } f(i) = s \text{ for all } i \in I \\ & & \quad \delta : [f_0]_\sim \mapsto f_0 \quad ) \end{array}$$

Figure 4: Extension template INDEX

The partitioned set of events $\Sigma_1$ remains unchanged. The extension is done by using the templates INJECT and INDEX applied in two consecutive steps. The first step injects states $S$ and values $V$ as products:

$$\text{INJECT } \mathcal{P}(S \times V) \text{ INTO } ((\Sigma_{in} \to P_1) \times (\Sigma_{out} \to P_1) \times \mathcal{PP}\Sigma_1)_\perp$$

with $r_0 = \{[\,]\}$ as the default value. In the second step, we index the occurrences of the process space on the right-hand side by values:

$$\text{INDEX } P_1 \text{ BY } V$$

with $t_0 = v_0 \mapsto p$ as default for a value $v_0$ and a given $p$. We can extend subparts of a type expression, such as $P_1$ here, without problems, if the subpart can be treated as a variable which is expanded.

The process descriptions $chaos, stop, e \mapsto p, p_1 [] p_2$ are extended using the canonical function lifting. The phrases $skip$, assignment, input $c?$ and output $c!a$ are newly introduced. Using the templates, behaviour preservation is guaranteed without discharging explicitly any proof obligation. The result of applying the templates to the function definitions is the following:

$$\begin{array}{lcl} \mathbf{P_2} & : & Process \to P_2 \\ \mathbf{P_2}[\![chaos]\!]s & \triangleq & \perp \\ \mathbf{P_2}[\![stop]\!]s & \triangleq & (\emptyset, [\,], [\,], \{(\emptyset, \emptyset, \emptyset)\}) \\ \mathbf{P_2}[\![e \to p]\!]s & \triangleq & \phi(\mathbf{P_1}[\![e \to p]\!]) \\ \mathbf{P_2}[\![p_1 [] p_2]\!]s & \triangleq & \text{let } (R_1, IN_1, OUT_1, S_1) = \mathbf{P_2}[\![p_1]\!]s, \\ & & \quad (R_2, IN_2, OUT_2, S_2) = \mathbf{P_2}[\![p_2]\!]s \text{ in} \\ & & \quad (R_1 \cup R_2, IN_1 []_m IN_2, OUT_1 []_m OUT_2, S_1 []_s S_2) \\ \mathbf{P_2}[\![skip]\!]s & \triangleq & (\{(s, ())\}, [\,], [\,], \{(\emptyset, \emptyset, \{\surd\})\}) \\ \mathbf{P_2}[\![x := e]\!]s & \triangleq & (\{(s \dagger [x \mapsto [\![e]\!]s], ())\}, [\,], [\,], \{(\emptyset, \emptyset, \{\surd\})\}) \\ \mathbf{P_2}[\![c?]\!]s & \triangleq & (\emptyset, [c \mapsto \lambda v.(\{(s, v)\}, [\,], [\,], \{(\emptyset, \emptyset, \{\surd\})\})], [\,], \{(\{c\}, \emptyset, \emptyset)\} \\ \mathbf{P_2}[\![c!a]\!]s & \triangleq & (\emptyset, [\,], [c \mapsto [[\![a]\!]s \mapsto (\{(s, ())\}, [\,], [\,], \{(\emptyset, \emptyset, \{\surd\})\})], \{(\emptyset, \{c\}, \emptyset)\} \end{array}$$

The formulation of $\mathbf{P_2}[\![p_1 [] p_2]\!]$ does not correspond directly to the application of the template, but is semantically equivalent and easier to read.

## 5.3 Extending the Domain Constraint

Again, we need to adapt the domain constraint to the new domain $P_2$. The domain constraint presented in section 4.3 is simply extended by adding:

$$\surd \in \bigcup \{\pi_3(z) | z \in S_1\} \quad \Leftrightarrow \quad \mathcal{P}(S \times V) \neq \emptyset$$

The constraint extension template is not used, since the two conditions from the first extension remain unchanged and another condition has been added explicitly. Since the resulting set for process state transformations, $\mathcal{P}(S \times V)$, to which the new constraint applies to, did not exist in the source language, the original constraint is preserved.

# 6 Related Work

Category theory and in particular monads (or Kleisli triples or just triples) are a recent, popular approach to modular description and integration of language features [Mog91, Wad92, LH96]. A (Kleisli) triple is a collection of extension operators on objects and morphisms of a category. A triple can describe a language feature, called notion of computation by Moggi, abstractly. Triples can be represented in categories. This corresponds to our semantics extensions and their representation in algebras. A recent attempt to use monads in the description and extension of specification languages is [CS97], where a customisable algebraic specification language is presented.

We have assumed a framework of sets and functions. This framework can be defined in terms of category theory by the category **Set** of sets and total functions. Category theory is a common mathematical framework to describe language semantics. We have formulated this paper in terms of universal algebra since RSL is defined using a classical denotational approach. A reformulation in terms of category theory is, though, possible.

The advantage of both approaches is that an abstract construct (our semantics extensions or monads) is provided which is a concise description of a feature and which can be represented in a number of concrete structures. The foundation of our semantics extensions is universal algebra, but a semantics extensions is a more specialised construct than a (Kleisli) triple. The equivalence is included which is used to model an observability criterion to express property preservation.

Refinement of software specifications and refinement of language descriptions share some properties. Both approaches to refinement are based on a notion of data refinement. In the area of language semantics, Riddle and Wallis [RW97] have presented ideas similar to ours. There, a refinement relation between denotationally specified languages is provided. The paper follows Schmidt's textbook on denotational semantics [Sch86]. Riddle and Wallis see definitions of semantic functions as semantic equations and define a correctness preserving refinement relation based on these equations.

Another algebraic approach can be found in [HJ98]. Algebraic theories are used to define a programming language by describing properties of language operators through predicates in the theory. Theories are linked e.g. in the process of refinement, expressed by functions. The subset relation is a straightforward relation. More general links include links between disjoint domains, e.g. based on Galois connections.

# 7 Conclusions

We have presented an extension framework for the modular extension of languages and their models. This framework allows a language designer to integrate denotationally specified language features. Our approach can be seen as part of a framework for formal language engineering, which aims at flexible combination and integration of specification languages. Languages can be customized for particular applications. The algebraic specification framework CoFI (Common Framework Initiative) [Gro99] is an example. Our framework shows that a number of common integration problems can be solved using an extension approach, i.e. starting with a basic language and adding or integrating new features step by step.

One of the important characteristics of our approach is the formulation of the extension in an abstract, modular way by using an operator called semantics extensions. Extension templates were introduced which allow us to discharge proof obligations regarding the preservation of properties automatically. A notation based on extension templates was provided to facilitate the use of the mathematical extension framework. The language designer is prevented from rewriting definitions in extensions.

RSL was used as a realistic, non-trivial case study to show the applicability of our approach. RSL is a typical example of a wide-spectrum specification language which made the integration of concurrency, state-based and modularisation features necessary. A full formal definition of a language is normally not feasible, but certain central aspects, such as the concurrency model for RSL and its connection to state-based features, can be investigated formally. Our framework can be applied in the design phase of an integrated specification language to get insight and understanding about the principles and mechanisms of the integration under investigation.

Extensions of the basic process model have already been presented in [Hen85], [HI93b], or [HI93a]. Using our framework, we could prove that properties such as behaviour and domain constraints are preserved if Hennessy's

acceptances model is extended by model-theoretic, state-based features to meet the requirement of the RAISE specification language[5]. We have dealt with this extension in a stepwise, modular way based on a selection of common extension templates. An approach of integrating concurrency and state-based features alternative to the one pursued by Bolignano and Debabi could have been taken. We could have started with an imperative model including states and values, on which processes are added and then events are distinguished.

We could extend our framework by introducing a metalanguage to express properties of programs and also properties of the extension. This creates a notation between the languages to be specified and the extension framework as presented so far. In [Pah98], we have presented a simple equational metalanguage. Extensions of this metalanguage could include a predefined equivalence symbol to interpret the behaviour preservation criterion.

Considering the results, and in particular [Pah97], it appears that the extensions here can be carried out in parallel, since they do not depend on each other. Extending in parallel means that a number of extensions are carried out on the same basic model resulting in a set of extended models. Under certain conditions, these extensions can be merged into one resulting model.

# References

[ACM96] ACM. *Workshop on Strategic Directions in Computing Research*. MIT, Cambridge, Massachusetts, USA, June 14-15, 1996. (http://www.acm.org/surveys/sdcr).

[BD92] D. Bolignano and M. Debabi. On the Foundations of the RAISE Specification Language Semantics. Technical report, ESPRIT project LACOS, 1992.

[Coh65] P.M. Cohn. *Universal Algebra*. Harper and Row Publishers, 1965.

[CS97] P. Cenciarelli and E. Saaman. Using Monads in Algebraic Specification. In *2th Workshop on Algebraic Development Technology (WADT97), Tarquinia*, 1997.

[Geo91] C. George. The RAISE Specification Language – A Tutorial. In S. Prehn and W.J. Toetenel, editors, *VDM'91 – Formal Software Development Methods*. Springer-Verlag, October 1991. LNCS 552.

[Grä68] G. Grätzer. *Universal Algebra*. D. van Nostrand Company, 1968.

[Gro92] The RAISE Language Group. *The RAISE Specification Language*. CRI A/S, Denmark, 1992.

[Gro99] The CoFI Working Group. CoFI: The Common Framework Initiative, http://www.brics.dk/Projects/CoFI/, 1999.

[Hen85] M. Hennessy. Acceptance Trees. *Journal of the ACM*, 32(4):896–928, October 1985.

[HI93a] M. Hennessy and A. Ingólfsdóttir. A Theory of Communicating Processes with Value-Passing. *Information and Computation*, 107(2), 1993.

[HI93b] M. Hennessy and A. Ingólfsdóttir. Communicating Processes with Value-passing Assignments. *Formal Aspects of Computing*, 3, 1993.

[HJ98] C.A.R. Hoare and H. Jifeng. *Unified Theories of Programming*. Prentice Hall, 1998.

[Hoa96] C.A.R. Hoare. Unified Theories of Programming. Working Material 3, International Summer School, Marktoberdorf, 1996.

[Jon90] C.B. Jones. *Systematic Software Development with VDM.* Prentice Hall, 1990.

[5] Passing processes via channels is also introduced by Bolignano and Debabi (p. 19-24 in [BD92]). The approach proposed involves a dependence of static and dynamic semantics. We have focussed here on dynamic semantics only.

[LH96] S. Liang and P. Hudak. Modular Denotational Semantics for Compiler Construction. In H.R. Nielson, editor, *Proceedings European Symposium on Programming ESOP'96*. Springer-Verlag, LNCS 1058, 1996.

[Mog91] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93:55–92, 1991.

[Pah97] C. Pahl. An Investigation into Parallel Extensions of the Unix C-shell Interpreter Language. Technical Report IT-TR:1997-015, Department of Information Technology, Technical University of Denmark, 1997.

[Pah98] C. Pahl. Facilitating Modular Property-Preserving Extensions of Programming Languages. In A. Butterfield and S. Flynn, editors, *Proc. 2nd Irish Workshop on Formal Methods, July 1998, Cork, Ireland*, Electronic Workshops in Computing. Springer-Verlag, 1998.

[RW97] S. Riddle and P. Wallis. Denotational Semantics and Refinement. In S. Flynn and G. O'Regan, editors, *Proc. 1st Irish Workshop on Formal Methods, July 1997, Dublin, Ireland*. Springer-Verlag, 1997.

[Sch86] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Wm.C. Brown Publishers, 1986.

[Wad92] P. Wadler. The essence of functional programming. In *Proc. 19th ACM Symp. on Principles of Programming Languages, Austin, Texas*, 1992. invited talk.